# [版本管理三国志 (CVS, Subversion, git)](http://www.cnblogs.com/vamei/archive/2013/02/21/2918069.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

最近有一则和git有关的新闻很火：

[12306的抢票插件拖垮了GitHub](http://www.zdnet.com/chinese-holiday-bookings-bring-down-hosting-site-7000010063/) (GitHub基于git)

git是一款版本控制软件(VCS，Version Control System)。VCS通常用于管理开发过程中的源代码文件。VCS是软件开发的好帮手。当软件本身在发布时获取大量关注时，VCS躲在幕后默默管理和记录软件的开发和发布进程。git颇有戏剧性的借春运抢票火了一把，也让许多人好奇什么是git，什么是VCS。我复习了一下VCS的历史，忽然有些读三国时的你方唱罢我登场的感觉，就想写一个VCS版本的三国志。

现在最常见的VCS软件(同时也是开源的VCS软件)有CVS, Subversion和git。CVS曾经雄霸一时，至今还管理着大量的开发项目。Subversion青出于蓝，对CVS进行改进，大有取而代之的势头。git另辟蹊径，依仗Linux的名号，并借GitHub的推广攻城略地。VCS领域激烈的争斗正反映了软件开发项目的红火势头。

### 斩白蛇而起

早期(1970年到1980年代)的软件开发大部分是愉快的个人创作。比如UNIX下的sed是L. E. McMahon写的，Python的第一个编译器是Guido写的，Linux最初的内核是Linus写的 (好吧，awk是个例外，它的名字是三位作者的首字母，但也只是三个人)。这些程序员可以用手工的方式进行备份，并以注释或者新建文本文件来记录变动。

正如现在普通用户常做的，当时的程序员常用cp备份:

$cp dev.c dev.bak

更有条理一些的程序员会加上一个时间标记，比如:

$cp dev.c dev.bak.19890908

程序员很可能会用vi创建一个LOG文件来做日志:

1989-09-08 02:00:00

Old input method is stupid

Add command-line input function

在一个版本发布的时候，程序员可能做一个tar归档，将所有的文件归为同一个.tar文件。

$tar -cf project\_v1.0.tar project

上面的工具构成了一套人工VCS。上面的这套组合也非常符合UNIX的模块化理念：让每个应用专注于一个小的功能，使用者根据需要，将这些功能连接起来。你还可以写一个shell脚本，将上面的功能都写在里面。当需要的时候，调用该脚本就可以了。

(这样一个shell脚本并不复杂，而且挺有用的，可以作为学习shell编程的小练习)

再说一下早期的合作开发模式。如在[Python简史](http://www.cnblogs.com/vamei/archive/2013/02/06/2892628.html%20)中看到的，Guido通过电子邮件接收补丁(patch)，并将补丁应用到原来的代码文件。实际上，一个补丁(patch)的主要功能是描述两个文件的改变(change, or file delta)。 假设我们有两个文件a.c和b.c内容分别为:  
a.c (有bug的代码)

int sum(int a, int b)

{

int c;

c = a + 1;

return c;

}

b.c (修正后的代码)

int sum(int a, int b)

{

int c;

c = a + b;

return c;

}

在UNIX系统下，运行

$diff a b > iss01.patch

iss01.patch就是一个补丁文件，它看起来如下：

4c4

< c = a + 1;

---

> c = a + b;

这个补丁表示，更改原文件第四行的c = a + 1;，改为c = a + b;，更改后的这一行位于新的文件的第四行。

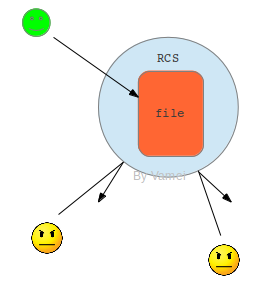
使用patch命令将iss01.patch应用到a.c上，相当于将 b.c-a.c 的改变作用在a上，a.c将和b.c有一样的内容：

$patch a.c < iss01.patch

当我发现a.c的代码有错误时，可以将我修改后的b.c与原来的a.c做diff获得补丁文件，并将补丁发给Guido，并告诉他该补丁是为了修正a.c代码中的加法错误。Guido确认之后，就可以使用patch应用该补丁了。在后面我们将看到，这种diff-patch的工作方式被VCS不同程度的采用。

### 东汉末年

早在70年代末80年代初，VCS的概念已经存在，比如UNIX平台的RCS (Revision Control System)。RCS是由Walter F. Tichy使用C开发。RCS对文件进行集中式管理，主要目的是避免多人合作情况下可能出现的冲突。如果多用户同时写入同一个文件，其写入结果可能相互混合和覆盖，从而造成结果的混乱。你可以将文件交给RCS管理。RCS允许多个用户同时读取文件，但只允许一个用户锁定(locking)并写入文件 (类似于[多线程的mutex](http://www.cnblogs.com/vamei/archive/2012/10/09/2715393.html%20))。这样，当一个程序员登出(check-out，见RCS的co命令)某个文件并对文件进行修改的时候。只有在这个程序完成修改，并登入(check-in，见RCS的ci命令)文件时，其他程序员才能登出文件。基本上RCS用户所需要的，就是co和ci两个命令。在co和ci之间，用户可以对原文件进行许多改变(change, or file delta)。一旦重新登入文件，这些改变将保存到RCS系统中。通过check-in将改变永久化的过程叫做提交(commit)。

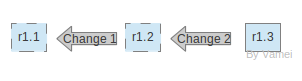


RCS互斥写入

RCS的互斥写入机制避免了多人同时修改同一个文件的可能，但代价是程序员长时间的等待，给团队合作带来不便。如果某个程序员登出了某个文件，而忘记登入，那他就要面对队友的怒火了。(从这个角度上来说，RCS造成的问题甚至大于它所解决的问题……)



文件每次commit都会创造一个新的版本(revision)。RCS给每个文件创建了一个追踪文档来记录版本的历史。这个文档的名字通常是原文件名加后缀,v (比如main.c的追踪文档为main.c,v)。追踪文档中包括：最新版本的文件内容，每次check-in的发生时间和用户，每次check-in发生的改变。在最新文档内容的基础上，减去历史上发生的改变，就可以恢复到之前的历史版本。这样，RCS就实现了备份历史和记录改变的功能。



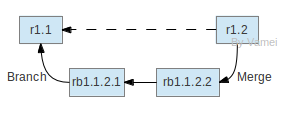
RCS历史版本追踪

相对与后来的版本管理软件，RCS纯粹线性的开发方式非常不利于团队合作。但RCS为多用户写入冲突提供了一种有效的解决方案。RCS的版本管理功能逐渐被其他软件(比如CVS)取代，但时至今日，它依然是常用的系统管理工具。RCS就像是东汉王室，飘摇多年而不倒。

### 挟天子，令诸侯

1986年，Dick Grune写了一系列的shell脚本用于版本管理，并最终以这些脚本为基础，构成了CVS (Concurrent Versions System)。CVS后来用C语言重写。CVS是开源软件。在当时，Stallman刚刚举起GNU的大旗，掀起开源允许的序幕。CVS被包含在GNU的软件包中，并因此得到广泛的推广，最终击败诸多商业版本的VCS，呈一统天下之势。

CVS继承了RCS的集中管理的理念。在CVS管理下的文件构成一个库(repository)。与RCS的锁定文件模式不同，CVS采用复制-修改-合并(copy-modify-merge)的模式，来实现多线开发。CVS引进了分支(branch)的概念。多个用户可以从主干(也就是中心库)创建分支。分支是主干文件在本地复制的副本。用户对本地副本进行修改。用户可以在分支提交(commit)多次修改。用户在分支的工作结束之后，需要将分支合并到主干中，以便让其他人看到自己的改动。所谓的合并，就是CVS将分支上发生的变化应用到主干的原文件上。比如下面的过程中，我们从r1.1分支出rb1.1.2.\*，并最终合并回主干，构成r1.2



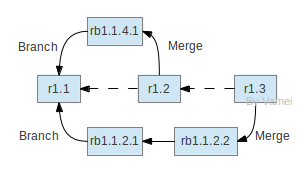
 copy-modify-merge

CVS与RCS类似，使用,v文件记录改变，以便追踪历史。在合并的过程中，CVS将两个change应用于r1.1，就得到了r1.2:

r1.2 = r1.1 + change(rb1.1.2.2 - rb1.1.2.1) + change(rb1.1.2.1-r1.1)

上面的两个改变都记录在,v文件中，所以很容易提取。

在多用户情况下，可以创建多个分支进行开发，比如：



在这样的多分支合并的情况下，有可能出现冲突(colliding)。比如上图中，第一次合并和第二次合并都对r1.1文件的同一行进行了修改，那么r1.3将不知道如何去修改这一行 (第二次合并比图示的要更复杂一些，分支需要先将主干拉到本地，合并过之后传回主干，但这一细节并不影响我们这里的讨论)。CVS要求冲突发生时的用户手动解决冲突。用户可以调用编辑器，对文件发生合并冲突的地方进行修改，以决定最终版本(r1.3)的内容。

CVS管理下的每个文件都有一系列独立的版本号(比如上面的r1.1,r1.2,r1.3)。但每个项目中往往包含有许多文件。CVS用标签(tag)来记录一个集合，这个集合中的元素是一对(文件名：版本号)。比如我们的项目中有三个文件(file1, file2, file3)，我们创建一个v1.0的标签：

tag v1.0 (file1:r1.3) (file2:r1.1) (file3:r1.5)

v1.0的tag中包括了r1.3版本的文件file1，r1.1版本的file2…… 一个项目在发布(release)的时候，往往要发布多个文件。标签可以用来记录该次发布的时候，是哪些版本的文件被发布。

CVS应用在许多重要的开源项目上。在90年代和00年代初，CVS在开源世界几乎不二选择 (RCS也是开源的，但正如我们已经提到的，RCS无法与CVS媲美)。CVS就像是官渡之战后的曹魏，挟开源运动，号令天下。时至今天，尽管CVS已经长达数年没有发布新版本，我们依然可以在许多项目中看到CVS的身影。

### 青出于蓝

正如曹操的统治富有争议一样(比如非汉祚，以臣欺君等等)，CVS也有许多常常被人诟病的地方，比如下面几条：

* 合并不是原子操作(atomic operation)：如果有两个用户同时合并，那么合并结果将是某种错乱的混合体。如果合并的过程中取消合并，不能撤销已经应用的改变。
* 文件的附加信息没有被追踪：一旦纳入CVS的管理，文件的附加信息(比如上次读取时间)就被固定了。CVS不追踪它所管理文件的附加信息的变化。
* 主要用于管理ASCII文件：不能方便的管理Binary文件和Unicode文件
* 分支与合并需要耗费大量的时间：CVS的分支和合并非常昂贵。分支需要复制，合并需要计算所有的改变并应用到主干。因此，CVS鼓励尽早合并分支。

CVS还有其它一些富有争议的地方。随着时间，人们对CVS的一些问题越来越感到不满 (而且程序员喜欢新鲜的东西)，Subversion应运而生。Subversion的开发者Karl Fogel和Jim Blandy是长期的CVS用户。赞助开发的CollabNet, Inc.希望他们写一个CVS的替代VCS。这个VCS应该有类似于CVS的工作方式，但对CVS的缺陷进行改进，并提供一些CVS缺失的功能。这就好像刘备从曹营拉出来单干的刘备一样。

总体上说，Subversion在许多方面沿袭CVS，也是集中管理库，通过记录改变来追踪历史，允许分支和合并，但并不鼓励过多分支。Subversion在一些方面得到改善。Subversion的合并是原子操作。它可以追踪文件的附加信息，并能够同样的管理Binary和Unicode文件。但CVS和Subversion又有许多不同：

* 与CVS的,v文件存储模式不同，Subversion采用关系型数据库来存储改变集。VCS相关数据变得不透明。
* CVS中的版本是针对某个文件的，CVS中每次commit生成一个文件的新版本。Subversion中的版本是针对整个文件系统的(包含多个文件以及文件组织方式)，每次commit生成一个整个项目文件系统树的新版本。

Subversion依赖类似于硬连接(hard link)的方式来提高效率，避免过多的复制文件本身。Subversion不会从库下载整个主干到本地，而只是下载主干的最新版本。

在Subversion刚刚诞生的时候，来自CVS用户的抱怨不断。他们觉得在Subversion中有太多的改动，有些改动甚至是相对于CVS的倒退。比如CVS中的tag，在Subversion中被改为直接复制版本的文件系统树到一个特殊的文件夹。然而，随着时间的推移，Subversion逐渐推广 (Subversion已经是Apache中自带的一个模块了，Subversion应用于GCC、SourceForge，新浪APP Engine等项目)，并依然有活跃的开发，而CVS则逐渐沉寂。事实上，许多UNIX的参考书的新版本中，都缩减甚至删除了CVS的内容。

### 别开生面

CVS和Subversion有很多不同的地方。但如果将这两者和git比较，那么git看起来就像孙权的碧眼，有一些怪异。

git的作者是Linus Torvald。对，就是写Linux Kernel的那个Linus Torvald。Linus在贡献了最初的Linux Kernel源代码之后，一直领导着Linux Kernel的开发。Linus Torvald本人相当厌恶CVS(以及Subversion)。然而，操作系统内核是复杂而庞大的代码“怪兽” （2012年的Linux Kernel有1500万行代码，Windows的代码不公开，估计远远超过这一数目）。Linux内核小组最初使用.tar文件来管理内核代码，但这远远无法匹配Linux内核代码的增长速度。Linus转而使用BitKeeper作为开发的VCS工具。BitKeeper是一款分布式的VCS工具，它可以快速的进行分支和合并。然而由于使用证书方面的争议(BitKeeper是闭源软件，但给Linux内核开发人员发放免费的使用证书)，Linus最终决定写一款开源的分布式VCS软件：git。

git在英文中比喻一个愚蠢或者不愉快的人(a stupid or unpleasant person)。Linus说这个比喻是在说自己：

I'm an egotistical bastard, and I name all my projects after myself. First "Linux", now "git".

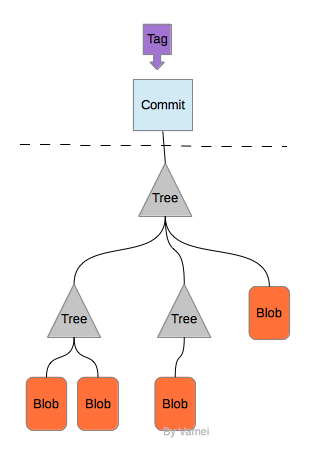
(这里，Linus似乎并不是在贬低自己，见Linus和Eric S. Raymond的争论: [The curse of the gifted](http://lwn.net/2000/0824/a/esr-sharing.php3))

对于一个开发项目，git会保存blob, tree, commit和tag四种对象。

* 文件被保存为blob对象。
* 文件夹被保存为tree对象。tree对象保存有指向文件或者其他tree对象指针。

上面两个对象类似于一个UNIX的文件系统，构成了一个文件系统树。

* 一个commit对象代表了某次提交，它保存有修改人，修改时间和附加信息，并指向一个文件树。这一点与Subversion类似，即每次提交为一个文件系统树。
* 一个tag对象包含有tag的名字，并指向一个commit对象。



虚线下面的对象构成了一个文件系统树。在git中，一次commit实际上就是一次对文件系统树的快照(snapshot)。

每个对象的内容的checksum校验(checksum校验可参阅[IP头部的checksum](http://www.cnblogs.com/vamei/archive/2012/12/02/2796988.html))都经过SHA1算法的HASH转换。每个对象都对应一个40个字符的HASH值。每个对象对应一个HASH值。两个内容不同的对象不会有相同的HASH值(SHA1有可能发生碰撞，但概率非常非常非常低)。这样，git可以随时识别各个对象。通过HASH值，我们可以知道这个对象是否发生改变。

比如一个文件LOG，它包含一下内容：

aaa

这个文件的HASH码为72943a16fb2c8f38f9dde202b7a70ccc19c52f34

如果我们修改这个文件，成为

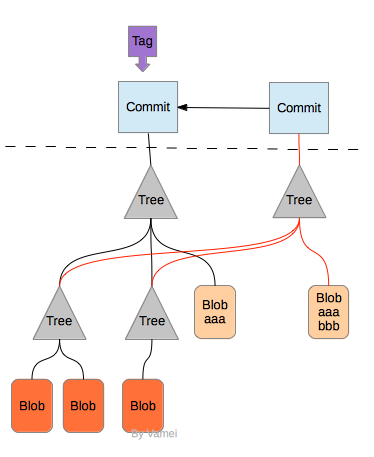
aaa

bbb

这个文件的HASH码变成dbee0265d31298531773537e6e37e4fd1ee71d62

所以，git只需看对象的HASH码，就可以知道该对象是否发生改变。

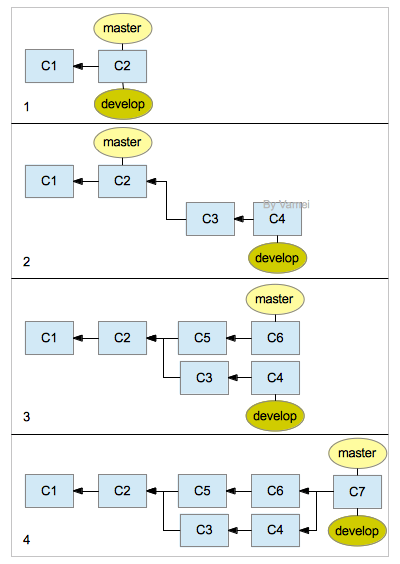
在整个开发过程中，可能会有许多次提交(commit)。每次commit的时候，git并不总是复制所有的对象。git会检验所有对象的HASH值。如果该对象的HASH值已经存在，说明该对象已经保存过，并且没有发生改变，所以git只需要调整新建tree或者commit中的指针，让它们指向已经保存过的对象就可以了。git在commit的时候，只会新建HASH值发生改变的对象。如下图所示，我们创建新的commit的时候，只需要新建一个commit对象，一个tree对象和一个blob对象就足够了，而不需要新建整个文件系统树。



可以看到，与CVS,Subversion保存改变(file delta)的方式形成对照，git保存的不是改变，而是此时的文件本身。由于不需要遵循改变路径来计算历史版本，所以git可以快速的查阅历史版本。git可以直接提取两个commit所保存的文件系统树，并迅速的算出两个commit之间的改变。

同样由于上面的数据结构，git可以很方便的创建分支(branch)。实际上，git的一个分支是一个指向某个commit的指针。合并时，git检查两个分支所指的两个commit，并找到它们共同的祖先commit。git会分别计算每个commit与祖先发生的改变，然后将两个改变合并(同样，针对同一行的两个改变可能发生冲突，需要手工解决冲突)。整个过程中，不需要复制和遵循路径计算总的改变，所以效率提高很多。

比如下面的图1中有两个分支，一个master和一个develop。我们先沿着develop分支工作，并进行了两次提交(比如修正bug1)，而master分支保持不变。随后沿着master分支,进行了两次提交(比如增加输入功能)，develop保持不变。在最终进行图4中的合并时，我们只需要将C4-C2和C6-C2的两个改变合并，并作用在C2上，就可以得到合并后的C7。合并之后，两个分支都指向C7。我们此时可以删除不需要的分支develop。



由于git创建、合并和删除分支的成本极为低廉，所以git鼓励根据需要创建多个分支。实际上，如果分支位于不同的站点(site)，属于不同的开发者，那么就构成了分布式的多分支开发模式。每个开发者都在本地复制有自己的库，并可以基于本地库创建多个本地分支工作。开发者可以在需要的时候，选取某个本地分支与远程分支合并。git可以方便的建立一个分布式的小型开发团队。比如我和朋友两人各有一个库，各自开发，并相互拉对方的库到本地库合并(如果上面master，develop代表了两个属于不同用户的分支，就代表了这一情况)。当然，git也允许集中式的公共仓库存在，或者多层的公共仓库，每个仓库享有不同的优先级。git的优势不在于引进了某种开发模式，而是给了你设计开发模式的自由。

正如东吴门阀合作的政治模式，git非集中式的开发模式让git成为了后起之秀。生子当如孙仲谋，生子当如Git Torvald。

(需要注意的是，GitHub尽管以git为核心，但并不是Linus创建的。事实上，[Linus不接收来自GitHub的Pull Request](https://github.com/torvalds/linux/pull/17#issuecomment-5654674)。Linus本人将此归罪于GitHub糟糕的Web UI。但有些搞笑的是，正是GitHub的Web页面让许多新手熟悉并开始使用git。好吧，Linus大婶是在鞭策GitHub。)

### 总结

和三国志不同，VCS的三国还没有决出最终胜负。或许Subversion会继续在一些重要项目上发挥作用，或许git会最终一统江山，或许CVS可以有新的发展并最终逆袭；又或许，一款新的VCS将取代所有的前辈。VCS激烈的竞争对于程序员来说是好事。一款优秀的VCS可以提高了我们管理项目的能力，降低我们犯错所可能支付的代价。随着开发项目越来越庞大和复杂，这一能力变得越来越不可缺少。花一点时间学习VCS，并习惯在工作中使用VCS，将会有意想不到的回报。

(我平时只用git，经验有限，如果有错漏，谢谢你的指正)